In [1]:
```
print('Hello World!')
```

```
Hello World!
```

In [2]:
```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
```

```
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __sizeof__(self, /)
|      Return the size of the string in memory, in bytes.
|
|  __str__(self, /)
|      Return str(self).
|
|  capitalize(self, /)
|      Return a capitalized version of the string.
|
|      More specifically, make the first character have upper case and the rest lower
|      case.
|
|  casefold(self, /)
|      Return a version of the string suitable for caseless comparisons.
|
|  center(self, width, fillchar=' ', /)
|      Return a centered string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  count(...)
|      S.count(sub[, start[, end]]) -> int
|
|      Return the number of non-overlapping occurrences of substring sub in
|      string S[start:end].  Optional arguments start and end are
|      interpreted as in slice notation.
|
|  encode(self, /, encoding='utf-8', errors='strict')
|      Encode the string using the codec registered for encoding.
|
|      encoding
|        The encoding in which to encode the string.
|      errors
|        The error handling scheme to use for encoding errors.
|        The default is 'strict' meaning that encoding errors raise a
|        UnicodeEncodeError.  Other possible values are 'ignore', 'replace' and
|        'xmlcharrefreplace' as well as any other name registered with
|        codecs.register_error that can handle UnicodeEncodeErrors.
|
|  endswith(...)
|      S.endswith(suffix[, start[, end]]) -> bool
|
|      Return True if S ends with the specified suffix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      suffix can also be a tuple of strings to try.
|
|  expandtabs(self, /, tabsize=8)
|      Return a copy where all tab characters are expanded using spaces.
|
```

```
|      If tabsize is not given, a tab size of 8 characters is assumed.
|
|  find(...)
|      S.find(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  format(...)
|      S.format(*args, **kwargs) -> str
|
|      Return a formatted version of S, using substitutions from args and kwargs.
|      The substitutions are identified by braces ('{' and '}').
|
|  format_map(...)
|      S.format_map(mapping) -> str
|
|      Return a formatted version of S, using substitutions from mapping.
|      The substitutions are identified by braces ('{' and '}').
|
|  index(...)
|      S.index(sub[, start[, end]]) -> int
|
|      Return the lowest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  isalnum(self, /)
|      Return True if the string is an alpha-numeric string, False otherwise.
|
|      A string is alpha-numeric if all characters in the string are alpha-numeric and
|      there is at least one character in the string.
|
|  isalpha(self, /)
|      Return True if the string is an alphabetic string, False otherwise.
|
|      A string is alphabetic if all characters in the string are alphabetic and there
|      is at least one character in the string.
|
|  isascii(self, /)
|      Return True if all characters in the string are ASCII, False otherwise.
|
|      ASCII characters have code points in the range U+0000-U+007F.
|      Empty string is ASCII too.
|
|  isdecimal(self, /)
|      Return True if the string is a decimal string, False otherwise.
|
|      A string is a decimal string if all characters in the string are decimal and
|      there is at least one character in the string.
|
|  isdigit(self, /)
|      Return True if the string is a digit string, False otherwise.
|
|      A string is a digit string if all characters in the string are digits and there
|      is at least one character in the string.
|
|  isidentifier(self, /)
```

```
|      Return True if the string is a valid Python identifier, False otherwise.
|
|      Use keyword.iskeyword() to test for reserved identifiers such as "def" and
|      "class".
|
|  islower(self, /)
|      Return True if the string is a lowercase string, False otherwise.
|
|      A string is lowercase if all cased characters in the string are lowercase and
|      there is at least one cased character in the string.
|
|  isnumeric(self, /)
|      Return True if the string is a numeric string, False otherwise.
|
|      A string is numeric if all characters in the string are numeric and there is at
|      least one character in the string.
|
|  isprintable(self, /)
|      Return True if the string is printable, False otherwise.
|
|      A string is printable if all of its characters are considered printable in
|      repr() or if it is empty.
|
|  isspace(self, /)
|      Return True if the string is a whitespace string, False otherwise.
|
|      A string is whitespace if all characters in the string are whitespace and there
|      is at least one character in the string.
|
|  istitle(self, /)
|      Return True if the string is a title-cased string, False otherwise.
|
|      In a title-cased string, upper- and title-case characters may only
|      follow uncased characters and lowercase characters only cased ones.
|
|  isupper(self, /)
|      Return True if the string is an uppercase string, False otherwise.
|
|      A string is uppercase if all cased characters in the string are uppercase and
|      there is at least one cased character in the string.
|
|  join(self, iterable, /)
|      Concatenate any number of strings.
|
|      The string whose method is called is inserted in between each given string.
|      The result is returned as a new string.
|
|      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
|
|  ljust(self, width, fillchar=' ', /)
|      Return a left-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  lower(self, /)
|      Return a copy of the string converted to lowercase.
|
|  lstrip(self, chars=None, /)
|      Return a copy of the string with leading whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  partition(self, sep, /)
```

```
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string.  If the separator is found,
|      returns a 3-tuple containing the part before the separator, the separator
|      itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing the original string
|      and two empty strings.
|
|  replace(self, old, new, count=-1, /)
|      Return a copy with all occurrences of substring old replaced by new.
|
|        count
|          Maximum number of occurrences to replace.
|          -1 (the default value) means replace all occurrences.
|
|      If the optional argument count is given, only the first count occurrences are
|      replaced.
|
|  rfind(...)
|      S.rfind(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Return -1 on failure.
|
|  rindex(...)
|      S.rindex(sub[, start[, end]]) -> int
|
|      Return the highest index in S where substring sub is found,
|      such that sub is contained within S[start:end].  Optional
|      arguments start and end are interpreted as in slice notation.
|
|      Raises ValueError when the substring is not found.
|
|  rjust(self, width, fillchar=' ', /)
|      Return a right-justified string of length width.
|
|      Padding is done using the specified fill character (default is a space).
|
|  rpartition(self, sep, /)
|      Partition the string into three parts using the given separator.
|
|      This will search for the separator in the string, starting at the end. If
|      the separator is found, returns a 3-tuple containing the part before the
|      separator, the separator itself, and the part after it.
|
|      If the separator is not found, returns a 3-tuple containing two empty strings
|      and the original string.
|
|  rsplit(self, /, sep=None, maxsplit=-1)
|      Return a list of the words in the string, using sep as the delimiter string.
|
|        sep
|          The delimiter according which to split the string.
|          None (the default value) means split according to any whitespace,
|          and discard empty strings from the result.
|        maxsplit
|          Maximum number of splits to do.
|          -1 (the default value) means no limit.
|
```

```
|      Splits are done starting at the end of the string and working to the front.
|
|  rstrip(self, chars=None, /)
|      Return a copy of the string with trailing whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  split(self, /, sep=None, maxsplit=-1)
|      Return a list of the words in the string, using sep as the delimiter string.
|
|      sep
|        The delimiter according which to split the string.
|        None (the default value) means split according to any whitespace,
|        and discard empty strings from the result.
|      maxsplit
|        Maximum number of splits to do.
|        -1 (the default value) means no limit.
|
|  splitlines(self, /, keepends=False)
|      Return a list of the lines in the string, breaking at line boundaries.
|
|      Line breaks are not included in the resulting list unless keepends is given and
|      true.
|
|  startswith(...)
|      S.startswith(prefix[, start[, end]]) -> bool
|
|      Return True if S starts with the specified prefix, False otherwise.
|      With optional start, test S beginning at that position.
|      With optional end, stop comparing S at that position.
|      prefix can also be a tuple of strings to try.
|
|  strip(self, chars=None, /)
|      Return a copy of the string with leading and trailing whitespace removed.
|
|      If chars is given and not None, remove characters in chars instead.
|
|  swapcase(self, /)
|      Convert uppercase characters to lowercase and lowercase characters to uppercase.
|
|  title(self, /)
|      Return a version of the string where each word is titlecased.
|
|      More specifically, words start with uppercased characters and all remaining
|      cased characters have lower case.
|
|  translate(self, table, /)
|      Replace each character in the string using the given translation table.
|
|        table
|          Translation table, which must be a mapping of Unicode ordinals to
|          Unicode ordinals, strings, or None.
|
|      The table must implement lookup/indexing via __getitem__, for instance a
|      dictionary or list.  If this operation raises LookupError, the character is
|      left untouched.  Characters mapped to None are deleted.
|
|  upper(self, /)
|      Return a copy of the string converted to uppercase.
|
|  zfill(self, width, /)
|      Pad a numeric string with zeros on the left, to fill a field of the given width.
|
```

```
|      The string is never truncated.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  maketrans(x, y=None, z=None, /)
|      Return a translation table usable for str.translate().
|
|      If there is only one argument, it must be a dictionary mapping Unicode
|      ordinals (integers) or characters to Unicode ordinals, strings or None.
|      Character keys will be then converted to ordinals.
|      If there are two arguments, they must be strings of equal length, and
|      in the resulting dictionary, each character in x will be mapped to the
```

In [3]:
```
massage = 'Hello World'
```

In [4]:
```
print(massage)
```

```
Hello World
```

In [5]:
```
massage='Bobby\'s World'
```

In [6]:
```
print(massage)
```

```
Bobby's World
```

In [7]:
```
massage="Bobby's World"
```

In [8]:
```
print(massage)
```

```
Bobby's World
```

Make multiple line string

In [9]:
```
massage = """I am Bangladeshi.
I love My country."""
```

In [10]:
```
massage
```

Out[10]:
```
'I am Bangladeshi. \nI love My country.'
```

In [11]:
```
massage = "Hello World"
```

len()

String length

In [12]:
```
print(len(massage))
```

```
11
```

Indexing

Syntax

string.index(value, start, end)

In [13]:
```
massage[0]      #printing string at 0 index
```

Out[13]: 'H'

In [14]:
```
massage[10]
```

Out[14]: 'd'

In [15]:
```
massage.index('W')
```

Out[15]: 6

In [16]:
```
"Mi casa, su casa.".rindex('casa')
```

Out[16]: 12

Slicing

In [17]:
```
massage[0:5]       #printing string in range 0 to 5(exclude 5 index)
```

Out[17]: 'Hello'

In [18]:
```
massage[:5]
```

Out[18]: 'Hello'

In [19]:
```
massage[6:]
```

Out[19]: 'World'

Negative Indexing

In [20]:
```
massage[-5:-1]
```

Out[20]: 'Worl'

lower()

In [21]:
```
massage.lower()
```

Out[21]: 'hello world'

upper()

In [22]:
```
massage.upper()
```

Out[22]: 'HELLO WORLD'

count()

In [23]:
```
massage.count("Hello")  #counting string or charecter in massage
```

Out[23]: 1

In [24]:
```
massage.count('l')
```

Out[24]: 3

find()

finding the string or character in which index

Syntax

string.find(value, start, end)

In [25]:
```
massage.find("World")
```

Out[25]: 6

In [26]:
```
massage.find('r')
```

Out[26]: 8

In [27]:
```
massage.find('Universe')    #if string not in massage returns -1
```

Out[27]: -1

In [28]:
```
"Mi casa, su casa.".rfind('casa')
```

Out[28]: 12

In [29]:
```
"Hello, welcome to my world.".rfind('e')
```

Out[29]: 13

In [30]:
```
"Hello, welcome to my world.".rfind('e',5,10)
```

Out[30]: 8

replace()

In [31]:
```
new_massage = massage.replace('World','Universe')
print(new_massage)
```

```
Hello Universe
```

In [32]:
```
"one one two one one was a race horse, two two was one too.".replace("one", "three", 3)
```

Out[32]: 'three three two three one was a race horse, two two was one too.'

String Concatenation

Adding string with + operator

In [33]:
```
greeting = 'Hello'
name = 'Rahat'
massage = greeting + ' ' + name + '. Welcome'
print(massage)
```

```
Hello Rahat. Welcome
```

f-strings – Formatted string literals

In [34]:
```
massage = f"{greeting.upper()} {name}. Welcome"
print(massage)
```

```
HELLO Rahat. Welcome
```

In [35]:
```
massage = "{} {}. Welcome".format(greeting,name.upper())
print(massage)
```

```
Hello RAHAT. Welcome
```

find all operation on a string

In [36]:
```
dir(name)
```

Out[36]:
```
['__add__',
 '__class__',
 '__contains__',
 '__delattr__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getitem__',
 '__getnewargs__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__iter__',
 '__le__',
 '__len__',
 '__lt__',
 '__mod__',
 '__mul__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__rmod__',
 '__rmul__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 'capitalize',
 'casefold',
 'center',
 'count',
 'encode',
 'endswith',
 'expandtabs',
 'find',
 'format',
 'format_map',
 'index',
 'isalnum',
 'isalpha',
 'isascii',
 'isdecimal',
 'isdigit',
 'isidentifier',
 'islower',
 'isnumeric',
 'isprintable',
 'isspace',
 'istitle',
 'isupper',
 'join',
 'ljust',
 'lower',
 'lstrip',
 'maketrans',
 'partition',
```

```
 'replace',
 'rfind',
 'rindex',
 'rjust',
 'rpartition',
 'rsplit',
 'rstrip',
 'split',
 'splitlines',
 'startswith',
 'strip',
 'swapcase',
 'title',
 'translate',
 'upper',
```

In [37]:
```
help(str.casefold)
```

```
Help on method_descriptor:

casefold(self, /)
    Return a version of the string suitable for caseless comparisons.
```

String Methods

strip()

The strip() method removes any whitespace from the beginning or the end

In [38]:
```
massage = " Hello, World! "
print(massage.strip())
```

```
Hello, World!
```

split()

The split() method splits the string into substrings if it finds instances of the separator

In [39]:
```
massage.split(",")
```

Out[39]:
```
[' Hello', ' World! ']
```

In [40]:
```
massage
```

Out[40]:
```
' Hello, World! '
```

Check String

To check if a certain phrase or character is present in a string, we can use the keywords in or not in

In [41]:
```
txt = "The rain in Spain stays mainly in the plain"
x = "ain" in txt
print(x)
```

```
True
```

In [42]:
```
"ain" not in txt
```

Out[42]:
```
False
```

Escape Character

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

In [43]:
```
"We are the so-called \"Vikings\" from the north."
```

Out[43]:
```
'We are the so-called "Vikings" from the north.'
```

Code Result

\' "Single Quote"

\ "Backslash"

\n "New Line"

\r "Carriage Return "

\t "Tab"

\b "Backspace"

\f "Form Feed"

\ooo "Octal value"

\xhh "Hex value"

capitalize()

The capitalize() method returns a string where the first character is upper case

In [44]:
```
txt = "hello, and welcome to my world."
```

In [45]:
```
txt.capitalize()
```

Out[45]:
```
'Hello, and welcome to my world.'
```

casefold()

The casefold() method returns a string where all the characters are lower case.

```
In [46]: txt="Hello, And Welcome To My World!"

In [47]: txt.casefold()
Out[47]: 'hello, and welcome to my world!'
```

center()

The center() method will center align the string, using a specified character (space is default) as the fill character.

Syntax

string.center(length, character)

```
In [48]: txt = "banana"

In [49]: txt.center(20)
Out[49]: '       banana       '

In [50]: txt.center(20,'.')
Out[50]: '.......banana.......'
```

encode()

The encode() method encodes the string, using the specified encoding. If no encoding is specified, UTF-8 will be used.

```
In [51]: txt = "My name is Ståle"

In [52]: txt.encode()
Out[52]: b'My name is St\xc3\xa5le'

In [53]: print(txt.encode(encoding="ascii",errors="backslashreplace"))
         print(txt.encode(encoding="ascii",errors="ignore"))
         print(txt.encode(encoding="ascii",errors="namereplace"))
         print(txt.encode(encoding="ascii",errors="replace"))
         print(txt.encode(encoding="ascii",errors="xmlcharrefreplace"))

         b'My name is St\\xe5le'
         b'My name is Stle'
         b'My name is St\\N{LATIN SMALL LETTER A WITH RING ABOVE}le'
         b'My name is St?le'
         b'My name is St&#229;le'
```

endswith()

The endswith() method returns True if the string ends with the specified value, otherwise False.

Syntax

string.endswith(value, start, end)

```
In [54]: txt = "Hello, welcome to my world."

In [55]: txt.endswith(".")
Out[55]: True

In [56]: txt.endswith("to",5,17)
Out[56]: True

In [57]: txt.endswith("my world.")
Out[57]: True

In [58]: txt.endswith("my world.", 5, 11)
Out[58]: False
```

expandtabs()

The expandtabs() method sets the tab size to the specified number of whitespaces.

Syntax

string.expandtabs(tabsize)

```
In [59]: txt = "H\te\tl\tl\to"

In [60]: txt.expandtabs(4)
Out[60]: 'H   e   l   l   o'

In [61]: print(txt)
         print(txt.expandtabs())
         print(txt.expandtabs(2))
         print(txt.expandtabs(4))
         print(txt.expandtabs(10))
         H       e       l       l       o
         H       e       l       l       o
         H e l l o
         H   e   l   l   o
         H         e         l         l         o
```

format()

The format() method formats the specified value(s) and insert them inside the string's placeholder.

Syntax

string.format(value1, value2...)

```
In [62]: txt = "For only {price:.2f} dollars"

In [63]: txt.format(price = 49)
Out[63]: 'For only 49.00 dollars'

In [64]: "My name is {fname}, I'am {age}".format(fname = "John", age = 36)
Out[64]: "My name is John, I'am 36"

In [65]: "My name is {0}, I'am {1}".format("John",36)
Out[65]: "My name is John, I'am 36"

In [66]: "My name is {}, I'am {}".format("John",36)
Out[66]: "My name is John, I'am 36"

In [67]: "My name is {1}, I am {0}".format(36,'John')
Out[67]: 'My name is John, I am 36'
```

Formatting Types

Inside the placeholders you can add a formatting type to format the result:

:< Left aligns the result (within the available space)

:> Right aligns the result (within the available space)

:^ Center aligns the result (within the available space)

:= Places the sign to the left most position

:+ Use a plus sign to indicate if the result is positive or negative

:- Use a minus sign for negative values only

: Use a space to insert an extra space before positive numbers (and a minus sign befor negative numbers)

:, Use a comma as a thousand separator

:_ Use a underscore as a thousand separator

:b Binary format

:c Converts the value into the corresponding unicode character

:d Decimal format

:e Scientific format, with a lower case e

:E Scientific format, with an upper case E

:f Fix point number format

:F Fix point number format, in uppercase format (show inf and nan as INF and NAN)

:g General format

:G General format (using a upper case E for scientific notations)

:o Octal format

:x Hex format, lower case

:X Hex format, upper case

:n Number format

:% Percentage format

In [68]:
```
txt = "You scored {:%}"
print(txt.format(0.25))

#Or, without any decimals:

txt = "You scored {:.0%}"
print(txt.format(0.25))
```

```
You scored 25.000000%
You scored 25%
```

isalnum()

The isalnum() method returns True if all the characters are alphanumeric, meaning alphabet letter (a-z) and numbers (0-9).

Example of characters that are not alphanumeric: (space)!#%&? etc.

In [69]:
```
"Company12".isalnum()
```

Out[69]: True

In [70]:
```
"Company,12".isalnum()
```

Out[70]: False

In [71]:
```
"Company 12".isalnum()
```

Out[71]: False

isalpha()

The isalpha() method returns True if all the characters are alphabet letters (a-z).

In [72]:
```
"Rahat".isalpha()
```

Out[72]: True

In [73]:
```
"Rahat42".isalpha()
```

Out[73]: False

isdecimal()

The isdecimal() method returns True if all the characters are decimals (0-9).

This method is used on unicode objects.

In [74]:
```
"\u0033".isdecimal()    #unicode for 3
```

Out[74]: True

```
In [75]: "\u0030".isdecimal()    #unicode for 0
Out[75]: True

In [76]: "\u0047".isdecimal()    #unicode for G
Out[76]: False

In [77]: "3".isdecimal()
Out[77]: True

In [78]: "0.3".isdecimal()
Out[78]: False
```

isdigit()

```
In [79]: "3456".isdigit()
Out[79]: True

In [80]: "\u00B2".isdigit()   #unicode for ²
Out[80]: True

In [81]: "X\u00B2"
Out[81]: 'X²'
```

isidentifier()

The isidentifier() method returns True if the string is a valid identifier, otherwise False.

A string is considered a valid identifier if it only contains alphanumeric letters (a-z) and (0-9), or underscores (_). A valid identifier cannot start with a number, or contain any spaces.

```
In [82]: "deMo".isidentifier()
Out[82]: True
```

In [83]:
```
a = "MyFolder"
b = "Demo002"
c = "2bring"
d = "my demo"

print(a.isidentifier())
print(b.isidentifier())
print(c.isidentifier())
print(d.isidentifier())
```

```
True
True
False
False
```

islower()

In [84]:
```
a = "Hello world!"
b = "hello 123"
c = "mynameisPeter"

print(a.islower())
print(b.islower())
print(c.islower())
```

```
False
True
False
```

isnumeric()

In [85]:
```
a = "\u0030" #unicode for 0
b = "\u00B2" #unicode for &sup2;
c = "10km2"

print(a.isnumeric())
print(b.isnumeric())
print(c.isnumeric())
```

```
True
True
False
```

In [86]:
```
"2343245".isnumeric()
```

Out[86]:
```
True
```

isprintable()

The isprintable() method returns True if all the characters are printable, otherwise False.

In [87]:
```
"Hello! Are you #1?".isprintable()
```

Out[87]: True

In [88]:
```
"Hello!\nAre you #1?".isprintable()
```

Out[88]: False

isspace()

The isspace() method returns True if all the characters in a string are whitespaces, otherwise False.

In [89]:
```
"    ".isspace()
```

Out[89]: True

In [90]:
```
"    a".isspace()
```

Out[90]: False

istitle()

The istitle() method returns True if all words in a text start with a upper case letter, AND the rest of the word are lower case letters, otherwise False.

In [91]:
```
"Hello, And Welcome To My World!".istitle()
```

Out[91]: True

In [92]:
```
"Hello, And Welcome To My world!".istitle()
```

Out[92]: False

In [93]:
```
a = "HELLO, AND WELCOME TO MY WORLD"
b = "Hello"
c = "22 Names"
d = "This Is %'!?"

print(a.istitle())
print(b.istitle())
print(c.istitle())
print(d.istitle())
```

```
False
True
True
True
```

isupper()

In [94]:
```
a = "Hello World!"
b = "hello 123"
c = "MY NAME IS PETER"

print(a.isupper())
print(b.isupper())
print(c.isupper())
```

```
False
False
True
```

join()

The join() method takes all items in an iterable and joins them into one string.

Syntax

string.join(iterable)

In [95]:
```
myTuple = ("John", "Peter", "Vicky")
```

In [96]:
```
"#".join(myTuple)
```

Out[96]:
```
'John#Peter#Vicky'
```

In [97]:
```
" ".join(myTuple)
```

Out[97]:
```
'John Peter Vicky'
```

In [98]:
```
"".join(myTuple)
```

Out[98]:
```
'JohnPeterVicky'
```

In [99]:
```
myDict = {"name": "John", "country": "Norway"}
mySeparator = "TEST"

x = mySeparator.join(myDict)

print(x)
```

```
nameTESTcountry
```

ljust() & rjust()

The ljust() method will left align the string, using a specified character (space is default) as the fill character.

In [100]:
```
"Rahat".ljust(10)+"is my name"
```

Out[100]:
```
'Rahat     is my name'
```

```
In [101]: "Rahat".ljust(10,'.')+"is my name"
Out[101]: 'Rahat.....is my name'

In [102]: "Rahat".rjust(10,'.')+" is my name"
Out[102]: '.....Rahat is my name'
```

strip()

The lstrip() method removes any leading characters (space is the default leading character to remove)

```
In [103]: ",,,,,ssaaww.....banana".lstrip(",.asw")
Out[103]: 'banana'

In [104]: ",,,,,ssaaww.....banana".strip(",.asw")
Out[104]: 'banan'

In [105]: ",,,,,ssaaww.....banana".rstrip(",.asw")
Out[105]: ',,,,,ssaaww.....banan'

In [106]: "   Rahat   ".strip()+" is my name"
Out[106]: 'Rahat is my name'

In [107]: ",,,,,rrttgg.....banana....rrr".strip(",.grt")
Out[107]: 'banana'
```

maketrans()

The maketrans() method returns a mapping table that can be used with the translate() method to replace specified characters.

Syntax

string.maketrans(x, y, z)

Parameter Values

Parameter Description

x -> Required. If only one parameter is specified, this has to be a dictionary describing how to perform the replace. If two or more parameters are specified, this parameter has to be a string specifying the characters you want to replace.

y -> Optional. A string with the same length as parameter x. Each character in the first parameter will be replaced with the corresponding character in this string.

z -> Optional. A string describing which characters to remove from the original string.

In [108]:
```
"Hello Sam".maketrans('S','R')
```

Out[108]:
```
{83: 82}
```

In [109]:
```
"Hello Sam".translate({83: 82})
```

Out[109]:
```
'Hello Ram'
```

In [110]:
```
"Hello Sam".translate("Hello Sam".maketrans('S','R'))
```

Out[110]:
```
'Hello Ram'
```

In [111]:
```
txt = "Good night Sam!";
x = "mSa";
y = "eJo";
z = "odnght";
mytable = txt.maketrans(x, y, z);
print(txt.maketrans(x, y, z));
print(txt.translate(mytable));
```

```
{109: 101, 83: 74, 97: 111, 111: None, 100: None, 110: None, 103: None, 104: None, 116: None}
G i Joe!
```

partition()

In [112]:
```
"I could eat bananas all day".partition('bananas')
```

Out[112]:
```
('I could eat ', 'bananas', ' all day')
```

In [113]:
```
"I could eat bananas all day".partition(' ')
```

Out[113]:
```
('I', ' ', 'could eat bananas all day')
```

In [114]:
```
"I could eat bananas all day".partition('apples')
```

Out[114]:
```
('I could eat bananas all day', '', '')
```

In [115]:
```
"I could eat bananas all day, bananas are my favorite fruit".rpartition('bananas')
```

Out[115]:
```
('I could eat bananas all day, ', 'bananas', ' are my favorite fruit')
```

split()

The split() method splits a string into a list.

Syntax

string.split(separator, maxsplit)

In [116]:
```
"My name is Rahat".split()
```

Out[116]:
```
['My', 'name', 'is', 'Rahat']
```

```
In [117]: "My name is Rahat".split(' ')
```

```
Out[117]: ['My', 'name', 'is', 'Rahat']
```

```
In [118]: 'apple,banana,pineapple,orange,goava,cherry'.split(',')
```

```
Out[118]: ['apple', 'banana', 'pineapple', 'orange', 'goava', 'cherry']
```

```
In [119]: 'apple,banana,pineapple,orange,goava,cherry'.split(',',3)
```

```
Out[119]: ['apple', 'banana', 'pineapple', 'orange,goava,cherry']
```

```
In [120]: 'apple,banana,pineapple,orange,goava,cherry'.rsplit(',',3)
```

```
Out[120]: ['apple,banana,pineapple', 'orange', 'goava', 'cherry']
```

splitlines()

```
In [121]: "Thank you for the music\nWelcome to the jungle".splitlines()
```

```
Out[121]: ['Thank you for the music', 'Welcome to the jungle']
```

```
In [122]: "Thank you for the music\nWelcome to the jungle".splitlines(True)
```

```
Out[122]: ['Thank you for the music\n', 'Welcome to the jungle']
```

startswith()

```
In [123]: "Hello, welcome to my world.".startswith('Hello')
```

```
Out[123]: True
```

```
In [124]: "Hello, welcome to my world.".startswith('wel',7,20)
```

```
Out[124]: True
```

swapcase()

```
In [125]: "Hello My Name Is PETER".swapcase()
```

```
Out[125]: 'hELLO mY nAME iS peter'
```

title()

```
In [126]: "Welcome to my world".title()
```

```
Out[126]: 'Welcome To My World'
```

In [127]:
```
"hello b2b2b2 and 3g3g3g".title()
```

Out[127]:
```
'Hello B2B2B2 And 3G3G3G'
```

zfill()

The zfill() method adds zeros (0) at the beginning of the string, until it reaches the specified length.

If the value of the len parameter is less than the length of the string, no filling is done.

Syntax

string.zfill(len)

In [128]:
```
a = "hello"
b = "welcome to the jungle"
c = "10.000"

print(a.zfill(10))
print(b.zfill(10))
print(c.zfill(10))
```

```
00000hello
welcome to the jungle
000010.000
```

In [129]:
```
help(str.zfill)
```

```
Help on method_descriptor:

zfill(self, width, /)
    Pad a numeric string with zeros on the left, to fill a field of the given width.

    The string is never truncated.
```